TÉCNICO LISBOA

Assignment 1

Parallel and Heterogeneous Computing Systems

Computação em Sistemas Paralelos e Heterogéneos (CSPH)

2021/2022

Overview

This assignment is intended to help you develop an understanding of the two primary forms of parallel execution present in a modern multi-core CPU:

1. Parallel execution using multiple cores (You'll see effects of Intel Hyper-threading as well.)
2. SIMD execution within a single processing core

You will also gain experience measuring and reasoning about the performance of parallel programs (a challenging, but important, skill you will use throughout this class). This assignment involves only a small amount of programming, but a lot of analysis!

Deadlines and Submission

You will submit Lab1 via Fenix (until 17th of October at 23h59, i.e., Sunday). The submission should be made in a single `zip` file with the following content:

- Your report (writeup) in a file called `report.pdf` (it must be pdf)
- Your implementation (codes) for <u>all</u> programs (please do `make clean` before submission)

When handed in, all codes must be compilable and runnable out of the box on the `cuda` machines! For problem 2, you need to fill all Excel sheets with your solution and hand it in together with the codes.

Your report should have <u>at most</u> 10 pages and should be well structured. For each problem (program) you should: elaborate your solution, explain your rationale (how did you arrive there, why it makes sense, describe all (if any) specific measurements/experiments that were conducted by you to prove your hypothesis etc). You should also discuss and analyze the obtained results!

Environment Setup

For this assignment, you will need to run (the final version of) your code on `cuda1` or `cuda2` machine (hostnames: `cuda1.scdeec.tecnico.ulisboa.pt` or `cuda2.scdeec.tecnico.ulisboa.pt` - we will abbreviate this as `{cuda1,cuda2}.scdeec.tecnico.ulisboa.pt`). You can freely use any `cuda` machine according to your preference (and their availability). Each `cuda` machine contains four-core 4.2 GHz Intel Core i7 processor (although dynamic frequency scaling can take it to 4.5 GHz when the chip decides it is useful and possible to do so). Each core in the processor can execute AVX2 vector instructions which describe simultaneous execution of the same operation on multiple single-precision data values. For the curious, a complete specification for this CPU can be found at https://ark.intel.com/.../intel-core-i7-7700k-processor-8m-cache-up-to-4-50-ghz.html[1]

The access to `cuda` machines is performed through `ssh`, by using the command:

```
ssh csph2021@{cuda1,cuda2}.scdeec.tecnico.ulisboa.pt
```

The password is `2021csph`.

[1] https://ark.intel.com/content/www/us/en/ark/products/97129/intel-core-i7-7700k-processor-8m-cache-up-to-4-50-ghz.html

Upon login, each group should create an individual folder by running the following command:

```
mkdir group_<group_number>
```

where you should substitute *<group_number>* with your group number (e.g., for the group number 3, the folder name should be `group_3`). You should develop your work only within your own folder! If in doubt, you can always find your group number on the course web-page (section Groups -> Labs (or any other group name that is listed) -> CTRL+F or CMD+F *to find your name or student number*).

IMPORTANT: Since the machines are shared resources, please verify if nobody else is performing experiments before running your program, in order not to affect the results of other groups. To check the users logged in the machine, you can use the **`who`** command. It is highly recommended to schedule the utilization of `cuda` machines among yourselves to avoid the simultaneous use of the machine (you should also think why it is not a good idea that all of you run the programs at the same time – given the characteristics of `cudas`' CPU – and how it can impact the execution of your program).

Note: For grading purposes, we expect you to report on the performance of code run on the `cuda` machine. However, for development, you may also want to run the programs in this assignment on your own machine. Feel free to include your findings from running code on other machines in your report as well, just be very clear what machine you were running on.

To get started:

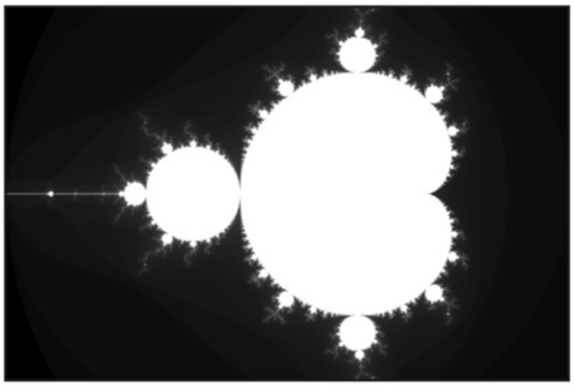
1. The assignment starter code is available on the course webpage (section: Labs). `wget` it, `unzip` it and start having fun with it (if you are using `cuda` machine, do this in your group folder!)

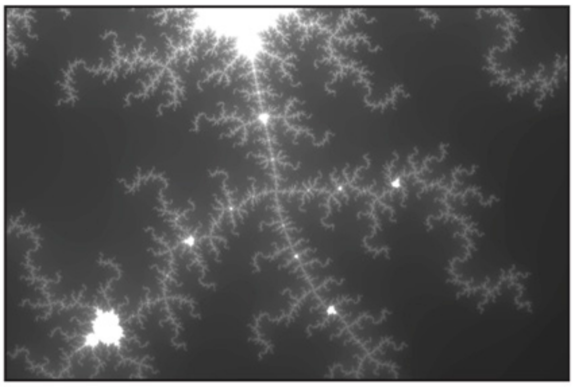
2. ISPC is needed to compile many of the programs used in this assignment. If you are using your own machine, you will first need to install the Intel SPMD Program Compiler (ISPC) available here: http://ispc.github.io/. ISPC can be easily installed on your Linux machine through the following steps (you don't need to this on cuda machines, since ISPC is already installed there):
   - Download the linux binary into a local directory of your choice. You can get ISPC compiler binaries for Linux from the ISPC downloads page[2]. You can use `wget` to directly download the binary from the downloads page:
     ```
     wget https://github.com/ispc/ispc/releases/download/v1.16.1/ispc-v1.16.1-linux.tar.gz
     ```
     Untar the downloaded file: `tar -xvf ispc-v1.16.1-linux.tar.gz`
     (You may need to change the ISPC version number in the filename accordingly.)
   - Add the ISPC directory to your system path. For example, if untarring the downloaded file produces the directory `~/Downloads/ispc-v1.14.1-linux`, in bash you'd update your path variable with:
     ```
     export PATH=$PATH:${HOME}/Downloads/ispc-v1.16.1-linux/bin
     ```
     The above line can be added to your `.bashrc` file for permanence. If you are using `csh`, you'll update your `PATH` using `setenv`. A quick Google search will teach you how.

[2] https://ispc.github.io/downloads.html

Program 1: Parallel Fractal Generation Using Threads

Build and run the code in the `prog1_mandelbrot_threads/` directory of the code base. (Type `make` to build, and `./mandelbrot` to run it.) This program produces the image file `mandelbrot-serial.ppm`, which is a visualization of a famous set of complex numbers called the Mandelbrot set. (Most platforms have a `.ppm` view. For example, to view the resulting images remotely, use `ssh -X` at the log in and the `display` command. Depending on your operating system, you might also need a X11 display server to use this command (like `XQuartz` for macOS, or `Xming` for Windows). As you can see in the images below, the result is a familiar and beautiful fractal. Each pixel in the image corresponds to a value in the complex plane, and the brightness of each pixel is proportional to the computational cost of determining whether the value is contained in the Mandelbrot set. To get image 2, use the command option `--view 2`. (See function `mandelbrotSerial()` defined in `mandelbrotSerial.cpp`). You can learn more about the definition of the Mandelbrot set at http://en.wikipedia.org/wiki/Mandelbrot_set.


View 1


View 2
(66x zoom)

Your job is to parallelize the computation of the images using std::thread (the "thing" we elaborated on our previous lecture). Starter code that spawns one additional thread is provided in the function `mandelbrotThread()` located in `mandelbrotThread.cpp`. In this function, the main application thread creates another additional thread using the constructor `std::thread(function, args...)`. It waits for this thread to complete by calling `join` on the thread object. Currently the launched thread does not do any computation and returns immediately. You should add code to `workerThreadStart` function to accomplish this task. You will not need to make use of any other `std::thread` API calls in this assignment.

What you need to do:

1. Modify the starter code to parallelize the Mandelbrot generation using two processors. Specifically, compute the top half of the image in thread 0, and the bottom half of the image in thread 1. This type of problem decomposition is referred to as *spatial decomposition* since different spatial regions of the image are computed by different processors.
2. Extend your code to use 2, 3, 4, 5, 6, 7, and 8 threads, partitioning the image generation work accordingly (threads should get blocks of the image). Note that the processor only has four cores but each core supports two hyper-threads, so it can execute a total of eight threads interleaved on its execution contents. In your write-up, produce a graph of **speedup compared**

to the reference sequential implementation as a function of the number of threads used **FOR VIEW 1**. Is speedup linear in the number of threads used? In your writeup hypothesize why this is (or is not) the case? (you may also wish to produce a graph for VIEW 2 to help you come up with a good answer. Hint: take a careful look at the three-thread datapoint.)

3. To confirm (or disprove) your hypothesis, measure the amount of time each thread requires to complete its work by inserting timing code at the beginning and end of `workerThreadStart()`. How do your measurements explain the speedup graph you previously created?
4. Modify the mapping of work to threads to improve speedup to at **about 7-8x on both views** of the Mandelbrot set (if you're around 7x that's fine, don't sweat it). You may not use any synchronization between threads in your solution. We are expecting you to come up with a single work decomposition policy that will work well for all thread counts – hard coding a solution specific to each configuration is not allowed! (Hint: There is a very simple static assignment that will achieve this goal, and no communication/synchronization among threads is necessary.). In your writeup, describe your approach to parallelization and report the final 8-thread speedup obtained.
5. Now run your improved code with 16 threads. Is performance noticeably greater than when running with eight threads? Why or why not?

Problem 2: Vectorizing Code using our "fake" SIMD Intrinsics

Take a look at the function `clampedExpSerial` provided below. The `clampedExp()` function raises `values[i]` to the power given by `exponents[i]` for all elements of the input array and clamps the resulting values at 9.999999. In Problem 2, your job is to vectorize this piece of code, so it can "run" on a machine with SIMD vector instructions.

However, rather than craft an implementation using SSE or AVX2 vector intrinsics that map to real SIMD vector instructions on modern CPUs, to make things a little easier, we're asking you to implement your version for our "fake processor" using our "fake vector intrinsics" from theoretical classes and run in on the paper. To be precise, in the Microsoft Excel sheets provided in `prog2_fake_intrinsics` folder.

The provided Excel file includes an example developed with our "fake" intrinsics, which determines the absolute value for each element of a vector (this example was heavily analyzed at the theoretical classes). This example also demonstrates how to use the file, as well as how to "scale" your code across "fake processors" with different SIMD widths (see `SIMD8`, `SIMD4`, `SIMD2`, and `SIMD16` sheets).

Here are few hints to help you in your implementation:

- Consider that the total number of loop iterations is always a multiple of SIMD vector width.
- *Hint:* For this exercise `_vpopcnt` (popcount) intrinsic is introduced in our "fake vector intrinsics". Popcount instructions count the number of set bits in a value, i.e., it returns a scalar value (integer). Using the `_vpopcnt` intrinsic on `__vbool` data may be helpful in this problem.

```
// For each element, compute values[i]^exponents[i] and clamp value to 9.999.
// Store result in output.
void clampedExpSerial(float* values, int* exponents, float* output, int N) {
  for (int i=0; i<N; i++) {
    float x = values[i];
    int y = exponents[i];
    if (y == 0) {
      output[i] = 1.f;
    } else {
      float result = x;
      int count = y - 1;
      while (count > 0) {
        result *= x;
        count--;
      }
      if (result > 9.999999f) {
        result = 9.999999f;
      }
      output[i] = result;
    }
  }
}
```

For the execution simulation in Excel, consider the following input values:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
values	3.0	2.0	2.5	1.25	5.5	0.5	10.1	3.15	1.75	6.55	1.63	1.5	4.33	0.15	1.95	2.83
exps.	0	2	3	10	0	4	0	3	5	1	4	9	0	5	0	3

You should consider the performance of your implementation to be the value "Total Vector Instructions" and "Execution Time [Cycles]". (Our Excel simulator assumes that every "fake" vector instruction or scalar instruction takes one cycle on the fake SIMD CPU.) Excel also calculates the "Vector Utilization", i.e., the percentage of vector lanes that are enabled during the execution of your code.

What you need to do:

1. Consider a CPU with the vector width of 8. Implement a vectorized version `clampedExpVector` of `clampedExpSerial` in the `SIMD8` sheet. Visually represent the execution of your code for the input values given above. Report the "Total Vector Instructions" and "Vector Utilization" metrics.
2. Consider that we change the vector width from 4, 8 to 16 (fill the corresponding Excel sheets). Does the vector utilization increase, decrease or stay the same as the vector width changes? Why? What about speedup (execution time)? Discuss those aspects in your write-up.

Program 3: Parallel Fractal Generation Using ISPC

Like Program 1, Program 3 computes a mandelbrot fractal image, but it achieves even greater speedups by utilizing both the CPU's four cores and the SIMD execution units within each core.

In Program 1, you parallelized image generation by creating one thread for each processing core in the system. Then, you assigned parts of the computation to each of these concurrently executing threads. (Since threads were one-to-one with processing cores in Program 1, you effectively assigned work explicitly to cores.) Instead of specifying a specific mapping of computations to concurrently executing threads, Program 3 uses ISPC language constructs to describe *independent computations*. These computations may be executed in parallel without violating program correctness (and indeed they will!). In the case of the Mandelbrot image, computing the value of each pixel is an independent computation. With this information, the ISPC compiler and runtime system take on the responsibility of generating a program that utilizes the CPU's collection of parallel execution resources as efficiently as possible.

You will make a simple fix to Program 3 which is written in a combination of C++ and ISPC (the error causes a performance problem, not a correctness one). With the correct fix, you should observe performance that is over thirty times greater than that of the original sequential Mandelbrot implementation from `mandelbrotSerial()`.

Program 3, Part 1. ISPC Basics

Before proceeding, you are encouraged to familiarize yourself with ISPC language constructs by reading through the ISPC walkthrough available at https://ispc.github.io/example.html. The example program in the walkthrough is almost exactly the same as Program 3's implementation of `mandelbrot_ispc()` in `mandelbrot.ispc`. In the assignment code, we have changed the bounds of the foreach loop to yield a more straightforward implementation.

What you need to do:

1. Compile and run the program `mandelbrot ispc`. **The ISPC compiler is currently configured to emit 8-wide AVX2 vector instructions.** What is the maximum speedup you expect given what you know about these CPUs? Why might the number you observe be less than this ideal? (Hint: Consider the characteristics of the computation you are performing? Describe the parts of the image that present challenges for SIMD execution? Comparing the performance of rendering the different views of the Mandelbrot set may help confirm your hypothesis.)

We remind you that for the code described in this subsection, the ISPC compiler maps gangs of program instances to SIMD instructions executed on a single core. This parallelization scheme differs from that of Program 1, where speedup was achieved by running threads on multiple cores.

Program 3, Part 2: ISPC Tasks

ISPCs SPMD execution model and mechanisms like `foreach` facilitate the creation of programs that utilize SIMD processing. The language also provides an additional mechanism utilizing multiple cores in an ISPC computation. This mechanism is launching *ISPC tasks*.

See the `launch[2]` command in the function `mandelbrot_ispc_withtasks`. This command launches two tasks. Each task defines a computation that will be executed by a gang of ISPC program instances.

As given by the function `mandelbrot_ispc_task`, each task computes a region of the final image. Similar to how the `foreach` construct defines loop iterations that can be carried out in any order (and in parallel) by ISPC program instances, the tasks created by this launch operation can be processed in any order (and in parallel on different CPU cores).

What you need to do:

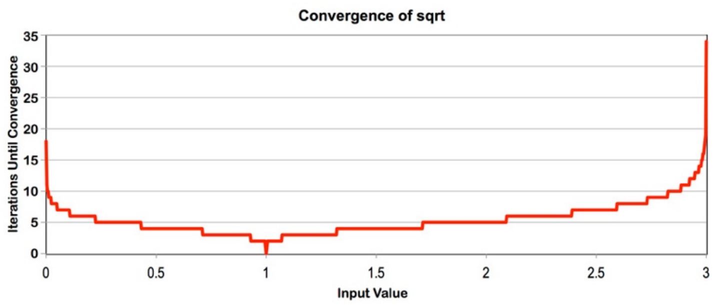
1. Run `mandelbrot_ispc` with the parameter `--tasks`. What speedup do you observe on view 1? What is the speedup over the version of mandelbrot_ispc that does not partition that computation into tasks?
2. There is a simple way to improve the performance of `mandelbrot_ispc --tasks` by changing the number of tasks the code creates. By only changing code in the function `mandelbrot_ispc_withtasks()`, you should be able to achieve performance that exceeds the sequential version of the code by over 30 times! How did you determine how many tasks to create? Why does the number you chose work best?

Program 4: Iterative sqrt

Program 4 is an ISPC program that computes the square root of 20 million random numbers between 0 and 3. It uses a fast, iterative implementation of square root that uses Newton's method to solve the equation $1/x^2 - s = 0$. This gives a value $x \approx \sqrt{1/s}$. Multiplying x by s gives an approximation to $\sqrt{s}$.

The value 1.0 is used as the initial guess in this implementation. The graph below shows the number of iterations required for sqrt to converge to an accurate solution for values in the (0-3) range. (The implementation does not converge for inputs outside this range). Notice that the speed of convergence depends on the accuracy of the initial guess.

Note: This problem is a review to double-check your understanding, as it covers similar concepts as programs 2 and 3.



What you need to do:

1. Build and run `sqrt`. Report the ISPC implementation speedup for single CPU core (no tasks) and when using all cores (with tasks). What is the speedup due to SIMD parallelization? What is the speedup due to multi-core parallelization?
2. Modify the contents of the array values to improve the relative speedup of the ISPC implementations. Describe a very-good-case input that **maximizes speedup over the sequential version of the code** and report the resulting speedup achieved (for both the with-

and without-tasks ISPC implementations). Does your modification improve SIMD speedup? Does it improve multi-core speedup (i.e., the benefit of moving from ISPC without-tasks to ISPC with tasks)? Please explain why.
3. Construct a very-bad-case input for sqrt that **minimizes speedup for ISPC with no tasks**. Describe this input, describe why you chose it, and report the resulting relative performance of the ISPC implementations. What is the reason for the loss in efficiency? **(keep in mind we are using the `--target=avx2` option for ISPC, which generates 8-wide SIMD instructions)**.

Program 5: BLAS saxpy

Program 5 is an implementation of the `saxpy` routine in the BLAS (Basic Linear Algebra Subproblems) library that is widely used (and heavily optimized) on many systems. `saxpy` computes the simple operation $\mathrm{result} = \mathrm{scale} * \mathrm{X} + \mathrm{Y}$, where X, Y, and result are vectors of N elements (in Program 5, N = 20 million) and scale is a scalar. Note that `saxpy` performs two math operations (one multiply, one add) for every three elements used. `saxpy` is a *trivially parallelizable computation* and features predictable, regular data access and predictable execution cost.

What you need to do:

1. Compile and run `saxpy`. The program will report the performance of ISPC (without tasks) and ISPC (with tasks) implementations of `saxpy`. What speedup from using ISPC with tasks do you observe? Explain the performance of this program. Do you think it can be substantially improved? (For example, could you rewrite the code to achieve near linear speedup? Yes or No? Please justify your answer.)

Notes: Don’t think too hard! We expect a simple answer, but the results from running this problem might trigger more questions in your head.